Problemas de *Vuelta Atrás*

Isabel Pita. Curso 20/21

Facultad de Informática - UCM

7 de enero de 2021

Bibliografía recomendada

- Estructuras de datos y métodos algorítmicos. 213 ejercicios resueltos. *Narciso Martí Oliet, Yolanda Ortega Mallén, y José Alberto Verdejo López*. Ibergaceta Publicaciones, segunda edición 2013.
- Estructuras de datos y métodos algorítmicos. Ejercicios resueltos. *Narciso Martí Oliet, Yolanda Ortega Mallén, y José Alberto Verdejo López*. Colección Prentice Práctica, Pearson Prentice-Hall, 2010.

Capítulo 14. Ejercicios resueltos: todos.

Objetivos

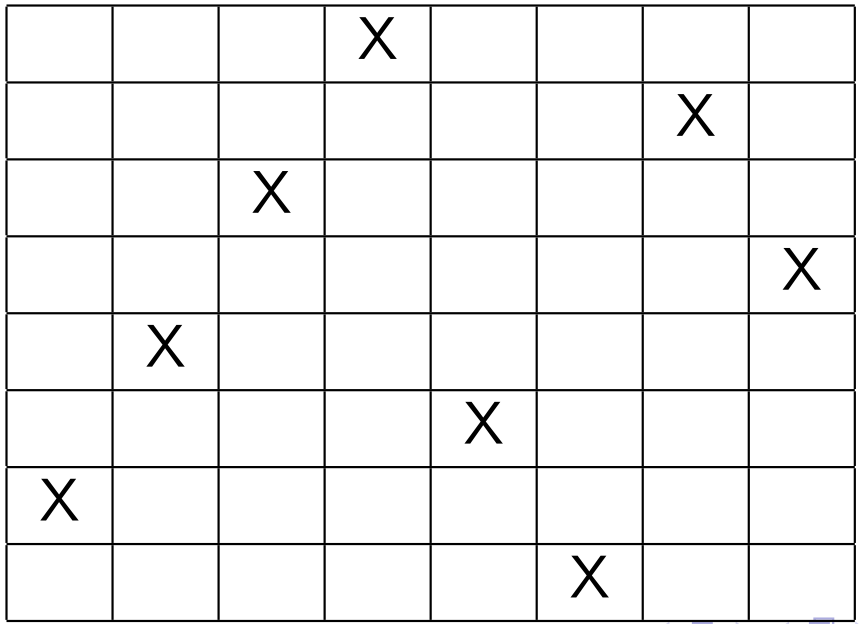
1. Esquema algorítmico de Vuelta atrás:
    1. Encontrar todas las soluciones a un problema
    2. Encontrar una solución
    3. Técnica de marcaje
    4. Encontrar la mejor solución

Problemas propuestos de acepta el reto

- 325. Helados de cucurucho
- 125. Números vampiros
- 220. Pasa la calculadora
- 240. Completa la suma

Problema de las *n*-reinas

- Colocar 8 reinas en un tablero de ajedrez sin que se amenacen. Dos reinas se amenazan si comparten la misma fila, columna o diagonal.
- Espacio de búsqueda: $\binom{64}{8} = 4{,}426{,}165{,}368$ soluciones.
- Solución: una 8-tupla $(x_1, .., x_8)$ en la que x_i representa la columna en la que se coloca la reina que está en la fila i-ésima del tablero.
- Ejemplo: La tupla $(4, 7, 3, 8, 2, 5, 1, 6)$ representa el tablero.



Las restricciones son:

- Restricciones explícitas para el problema de las reinas:
  - $S_i = \{1, \ldots, 8\}, 1 \leq i \leq 8$. Es decir, cada columna tiene que estar dentro del tablero.
  - Esta representación hace que el espacio de soluciones potenciales se reduzca a 8^8 posibilidades (16.777.216 valores).
- Restricciones implícitas para el problema de las reinas:
  - No puede haber dos reinas en la misma columna, ni en la misma diagonal.
  - Al no poder haber dos reinas en la misma columna, se deduce que todas las soluciones son permutaciones de la 8-tupla $(1, 2, 3, 4, 5, 6, 7, 8)$. Por lo tanto el espacio de soluciones potenciales se reduce a 8! (40.320 valores diferentes).

Algoritmo.

```
void nReinas(vector<int> & sol, int k){
    for(int i = 0; i < sol.size(); ++i){
        sol[k] = i;
        if (esValida(sol, k)){
            if(k == sol.size() - 1){
                tratarSolucion(k, sol.size());
            }
            else{
                nReinas(sol, k + 1);
            }
        }
    }
}
```

- `esValida` comprueba que la nueva reina no amenaza a las anteriores.

```
bool esValida(vector<int> const& sol, int k) {
    bool correcto = true;
    int i = 0;
    while (i < k && correcto){
        if(sol[i] == sol[k]
            || abs(sol[k] - sol[i]) == k - i))
            correcto = false;
        ++i;
    }
    return correcto;
}
```

- `solucion[i] == solucion[k]`: la nueva reina está en la misma columna que las anteriores,
- `abs(solucion[k] - solucion[i]) == k - i)`: comparten diagonal,
- Nunca puede estar en la misma fila por la manera en que construimos la solución.

- `tratarSolucion` escribe la solución en la salida:

```
void tratarSolucion(vector<int> const& sol){
    cout << "Solucion: ";
    for(int i : sol) cout << i << ' ';
    cout << '\n';
}
```

- Llamada inicial:

```
void nReinas(int n){
    vector<int> solucion(n);
    nReinas(solucion, 0);
}

int main()
{
    nReinas(8);
    return 0;
}
```

Optimizaciones al problema de las reinas

- Las soluciones son simétricas. Si hay una solución colocando la primera reina en la casilla 2 también la habrá colocando la reina inicial en la casilla $n-2$. Reducir el espacio de búsqueda a la mitad ejecutando el método recursivo para el primer nivel sólo para las casillas menores de $n/2$.

Optimizaciones

- `esValida` tiene un coste lineal en función de la entrada. Utilizar la técnica de marcaje.
  - Primera aproximación: Utilizar como *marca* una estructura de datos *tablero*.
    - Al añadir una reina *se marcan* las casillas amenazadas por la nueva reina.
    - Para comprobar si una nueva reina está amenazada se comprueba si esta marcada la casilla correspondiente del tablero.
    - Problema: marcar en el tablero las casilla que amenaza la nueva reina supone un coste lineal
  - Segunda aproximación: utilizar dos vectores: uno con las columnas amenazadas y otro con las diagonales amenazadas.
    - El vector de columnas tiene tamaño n,
    - El número de diagonales es: $4n - 2$.
    - La modificación y acceso a ambos vectores tiene coste constante.
    - El problema esta resuelto en detalle en el capítulo 14 del libro (Martí Oliet et al., 2013).

Ejemplo de búsqueda de una sola solución: Dominó

Se trata de encontrar una cadena circular de fichas de dominó.
Teniendo en cuenta:

- Cada cadena tiene que utilizar las 28 fichas diferentes que contiene el juego de dominó.
- No se puede repetir ninguna ficha.
- Las cadenas tienen que ser correctas, es decir, cada ficha tiene que ser compatible con la siguiente y la cadena tiene que cerrar (el valor de un extremo de la última ficha tiene que coincidir con el otro extremo de la primera). Por ejemplo: $6|3 \rightarrow 3|4 \rightarrow 4|1 \rightarrow 1|0 \rightarrow .. \rightarrow 5|6$ es una cadena correcta.

- Solución: tupla de 29 valores $(x_0, .., x_{28})$ cada x_i es un número del 0 al 6, que representa el valor de uno de los extremos de la ficha.
- Se declara una posición más en la tupla para poder realizar la comprobación de que la cadena es cerrada.
- Para evitar fichas repetidas utilizaremos una matriz (7×7) donde marcaremos las fichas usadas. Al marcar la casilla (i, j), hay que marcar la simétrica (j, i), ya que se trata de la misma ficha.
- El problema pide que se encuentre una sola solución. Utilizaremos una variable de control `exito` para abortar la búsqueda cuando se encuentre la primera solución.

```
void domino (vector<int> & sol, int k, int n,
    vector<vector<bool>> & marcas, bool &exito){
    int i = 0;  int m = (n * n + n) /2;
    while (i < n && !exito){
        if(!marcas[sol[k-1]][i]){
            sol[k] = i;
            if(k == m) {
              if (sol[0] == sol[k]){
                tratarSolucion(sol,m);
                exito = true;
              }
            }
            else {
                marcas[sol[k-1]][i] = true;
                marcas[i][sol[k-1]] = true;
                domino(sol, k + 1, n, marcas, exito);
                marcas[sol[k-1]][i] = false;
                marcas[i][sol[k-1]] = false;
           }
        }
        ++i;   }}
```

- Llamada inicial:

```
int main() {
    vector<int> sol(29);
    vector<vector<bool>> marcas(7,vector<bool>(7));
    for(int i = 0; i < 7; ++i)
        for(int j = 0; j < 7; ++j)
            marcas[i][j] = false;

    sol[0] = 6;  sol[1] = 6;
    marcas[6][6] = true;
    bool exito = false;
    domino(sol, 2, 7, marcas, exito);
    return 0;
}
```

- El número de valores posibles de las fichas es $n = 6$. La matriz de marcas se declara de dimensión $n \times n$ y el vector solución es de tamaño $(n \times n + n)/2 + 1$.
- Empezamos por el doble 6. Se podría haber utilizado cualquier par de valores.

Optimización

- Para obtener la *mejor* solución tenemos que almacenar la mejor solución hasta el momento. Cada nueva solución se compara con la que tenemos almacenada.
- Problema del viajante (*Travelling Salesman Problem*)
  Sean N ciudades. El objetivo es encontrar una ruta que, comenzando y terminando en una ciudad concreta, pase una sola vez por cada una de las ciudades intermedias y minimice la distancia recorrida por el viajante.

- Hay que encontrar una permutación del conjunto de ciudades $P = \{c_0, .., c_N\}$ tal que: $\sum_{i:0..N-1} d[c_i, c_{(i+1)\,\%N}]$ sea mínimo.
- La distancia d entre dos ciudades viene dada en una matriz .
- El tamaño del árbol de exploración es $(N-1)!$, ya que corresponde a todas las posibles permutaciones, teniendo en cuenta que el principio y el final es el mismo.
- Aplicaciones prácticas:
  - área de logística de transporte.
  - Robótica, minimizar el número de desplazamientos al realizar una serie de perforaciones en una plancha o en un circuito impreso.
  - Control y operativa optimizada de semáforos, etc.

```
void viajante(vector<vector<int>> const& d,
        vector<int>& sol, int coste, int k,
        int n,vector<int>& solMejor, int &cMejor) {
    for(int i = 0; i < n; ++i){
        sol[k] = i;
        coste += d[sol[k-1]][sol[k]];
        if(esValida(sol, k)){
            if(esSolucion(sol,k)) {
                if(coste < cMejor) {
                    cMejor = coste;
                    copiarSolucion(sol, solMejor);
                }
            }
            else viajante(d, sol, coste, k+1, n,
                          solMejor, cMejor);
        }
        coste -= d[sol[k-1]][sol[k]];
    }
}
```

- Mejorar la búsqueda del camino óptimo utilizando una estimación optimista para realizar podas tempranas.
- Idea: prever el mínimo coste de lo que falta por recorrer. Si ese coste, sumado al que llevamos acumulado, supera la mejor solución encontrada hasta el momento $\Rightarrow$ abandonar la búsqueda.
- Cálculo de una estimación optimista: encontrar la mínima distancia entre cualquier par de ciudades, y considerar que todos los desplazamientos van a tener esa distancia.
- Se pueden realizar cálculos más ajustados del coste del camino que queda por recorrer, pero hay que tener en cuenta que el cálculo debe ser sencillo para no aumentar el coste del algoritmo.

- Añadir antes de realizar la llamada recursiva, el cálculo del coste estimado.

```
int costeEst = coste + (n - k ) * costeMin;
if(costeEst < cMejor)
   viajante(d,sol, coste, k+1, n, solMejor, cMejor);
```

- El `costeMinimo` se calcula muy fácilmente recorriendo la matriz de distancias; consideramos que se ha procesado al principio de la ejecución y que lo tenemos almacenado en un parámetro.
- Solo realizaremos la recursión si la solución se puede mejorar.
- Para mejorar el coste de la función `esValida` utilizar la técnica de marcaje. Se declara un vector `usado` de n componentes, donde el valor de cada componente indica si la ciudad correspondiente ha sido visitada.

Problema de la mochila

- Tenemos n objetos con valor $(v_0, .., v_{n-1})$ y peso $(p_0, .., p_{n-1})$, y queremos determinar qué objetos transportar en la mochila sin superar su capacidad m (en peso) para maximizar el valor del contenido de la mochila.
- La solución es una tupla $(b_0, .., b_{n-1})$ de booleanos, cada b_1 indica si se coge el i-ésimo elemento.
- Restricciones:
  - Debemos maximizar el valor de lo que llevamos $\sum_{i:0..n-1} b_i v_i$.
  - El peso no debe exceder el máximo permitido $\sum_{i:0..n-1} b_i p_i \leq m$.

```
void mochila(vector<float> const& P,
     vector<float> const& V, vector<bool>& sol,
     int k, int n, int m, float peso, float beneficio,
     vector<int>& solMejor, int &valorMejor)  {
   // hijo izquierdo [cogemos el objeto]
   sol[k] = true;
   peso = peso + P[k];
   beneficio = beneficio + V[k];
   if(peso <= m) {
        if(k == n-1) {
           if(valorMejor < beneficio){
               valorMejor = beneficio;
               copiarSolucion(sol, solMejor);
            }
         }
         else {
             mochila(P,V,sol, k+1,n, m, peso, beneficio,
                              solMejor, valorMejor);
          }
   }
```

```
    peso = peso - P[k];   //desmarcamos peso y beneficio
    beneficio = beneficio - V[k];
    // hijo derecho [no cogemos el objeto]
    sol[k] = false;
    if(k == n-1){
       if(valorMejor < beneficio)){
          valorMejor = beneficio;
          copiarSolucion(sol, solMejor);
       }
    }
    else{
       mochila(P,V,sol,  k+1,n, m,peso, beneficio,
       solMejor, valorMejor);
  }

}
```

En el hijo de la derecha no tendremos que comprobar si excedemos el peso total, ya que al descartar el objeto no aumentamos el peso acumulado.

Optimización problema de la mochila

- Cáculo de una cota superior (una evaluación optimista) del beneficio que podemos obtener con lo que nos resta para rellenar la mochila.
  - Organizamos inicialmente los objetos en los vectores `P` y `V` de manera que estén ordenados por "densidad de valor" decreciente. Llamamos densidad de valor al cociente v_i/p_i.
  - Cogeremos primero los objetos que tienen más valor por unidad de peso. Si al decidir sobre el objeto k hemos alcanzado un beneficio `b` y un peso `p`, estimamos el beneficio optimista como la suma de `b` más el beneficio conseguido cogiendo los objetos que quepan en el orden indicado desde el $k+1$ al $n-1$.
  - Si se llega a un objeto j que ya no cabe, se fracciona y se suma el valor de la fracción que quepa. (Solución *voraz*, produce siempre una cota superior a cualquier solución donde no se permita faccionamiento).
  - La poda se produce si el beneficio optimista es **menor** que el beneficio de la mejor solución alcanzada hasta el momento.

Coloreado de mapas

- Si el mapa M tiene m países, numerados 0 a $m-1$, entonces la solución va a ser una tupla $(x_0, .., x_{m-1})$ donde x_i es el color asignado al i-ésimo pais.
- Cada elemento x_i de la tupla pertenecerá al conjunto $\{0, .., n-1\}$ de colores válidos.

Cada vez que vayamos a pintar un país de un color tendremos que comprobar que ninguno de los adyacentes está pintado con el mismo color. En este caso es más sencillo hacer la comprobación cada vez que coloreamos un vértice en lugar de utilizar *marcaje*.

```
void colorear(vector<int> &sol, int k, int n){
    for(int c = 0; c < n; c++){
        sol[k] = c;
        if(esValida(sol, k)){
            if(esSolucion(k,sol.size())){
                tratarSolucion(sol);
            }
            else{
               colorear(sol, k + 1, n);
            }
        }
    }
}
```

`esValida` comprueba si la solución parcial cumple que dos países limítrofes no compartan color; Asumimos que tenemos acceso a cierto objeto `M` donde se guarda el mapa, y que tiene un método que dice si dos países son fronterizos.

```
bool esValida(vector<int> const& sol, int k) {
  int i = 0; bool valida = true;
  while (i < k && valida) {
    if (M.hayFrontera(i, k)  && sol[k] == sol[i])
      valida =  false;
    i++;
    }
  return valida;
}
```

A la hora de hacer la llamada inicial podemos asignar al primer país del mapa un color arbitrario.